IN THE UNITED STATES PATENT AND TRADEMARK OFFICE
Before the Board of Patent Appeals and Interferences

In re Patent Application of
NEVILL
Serial No. 09/887,561
Filed: June 25, 2001
Title: INTERCALLING BETWEEN NATIVE AND NON-NATIVE INSTRUCTION SETS

Atty Dkt. 550-244
C# M#
TC/A.U.: 2191
Examiner: Tang, K.
Date: January 30, 2006

Mail Stop Appeal Brief - Patents
Commissioner for Patents
P.O. Box 1450
Alexandria, VA 22313-1450

Sir:

☐ **Correspondence Address Indication Form Attached.**

☐ **NOTICE OF APPEAL**
Applicant hereby **appeals** to the Board of Patent Appeals and Interferences from the last decision of the Examiner twice/finally rejecting applicant's claim(s). $500.00 (1401)/$250.00 (2401) $

☒ An appeal **BRIEF** is attached in the pending appeal of the above-identified application $500.00 (1402)/$250.00 (2402) $ 500.00

☐ Credit for fees paid in prior appeal without decision on merits -$ ( )

☐ A reply brief is attached. **(no fee)**

☒ Petition is hereby made to extend the current due date so as to cover the filing date of this paper and attachment(s)
One Month Extension $120.00 (1251)/$60.00 (2251)
Two Month Extensions $450.00 (1252)/$225.00 (2252)
Three Month Extensions $1020.00 (1253/$510.00 (2253)
Four Month Extensions $1590.00 (1254/$795.00 (2254) $ 1590.00

☐ "Small entity" statement attached.

Less month extension previously paid on -$( )

TOTAL FEE ENCLOSED $ 2090.00

Any future submission requiring an extension of time is hereby stated to include a petition for such time extension. The Commissioner is hereby authorized to charge any deficiency, or credit any overpayment, in the fee(s) filed, or asserted to be filed, or which should have been filed herewith (or with any paper hereafter filed in this application by this firm) to our **Account No. 14-1140.** A duplicate copy of this sheet is attached.

901 North Glebe Road, 11th Floor
Arlington, Virginia 22203-1808
Telephone: (703) 816-4000
Facsimile: (703) 816-4100
JRL:sd

NIXON & VANDERHYE P.C.
By Atty: John R. Lastova, Reg. No. 33,149

Signature: John R. Lastova

01/31/2006 SZEWDIE1 00000024 09887561

~~01 FC:1402~~ ~~500.00 OP~~
02 FC:1254 1590.00 OP

IN THE UNITED STATES PATENT AND TRADEMARK OFFICE

In re Patent Application of

NEVILL — Atty. Ref.: 550-244

Serial No. 09/887,561 — Group: 2191

Filed: June 25, 2001 — Examiner: Tang, K.

For: INTERCALLING BETWEEN NATIVE AND NON-NATIVE INSTRUCTION SETS

Before the Board of Patent Appeals and Interferences

BRIEF FOR APPELLANT

On Appeal From Final Rejection
From Group Art Unit 2191

John R. Lastova
NIXON & VANDERHYE P.C.
11th Floor, 901 North Glebe Road
Arlington, Virginia 22203-1808
(703) 816-4025
Attorney for Appellants
Nevill
ARM Limited

IN THE UNITED STATES PATENT AND TRADEMARK OFFICE
BEFORE THE BOARD OF PATENT APPEALS AND INTERFERENCES

In re Patent Application of

NEVILL — Atty. Ref.: 550-244

Serial No. 09/887,561 — Group: 2191

Filed: June 25, 2001 — Examiner: Tang, K.

For: INTERCALLING BETWEEN NATIVE AND NON-NATIVE INSTRUCTION SETS

* *

January 30, 2006

Mail Stop Appeal Brief - Patents
Commissioner for Patents
P.O. Box 1450
Alexandria, VA 22313-1450

APPEAL BRIEF

I. REAL PARTY IN INTEREST

The real party in interest is the assignee, ARM Limited, a United Kingdom corporation.

II. RELATED APPEALS AND INTERFERENCES

There are no other appeals related to this subject application. There are no interferences related to this subject application.

III. STATUS OF CLAIMS

Claims 1-16 and 20-28 are pending. Claims 1-5, 15-16, and 20 stand rejected under 35 U.S.C. §102(b) as being anticipated by Guccione. Claims 6-9 and 21-24 stand

rejected under 35 U.S.C. §103 for obviousness over Guccione in view of Yates (6,091,897). Claims 10-14 and 25-28 stand rejected under 35 U.S.C. §103 for obviousness over Guccione in view of Weiss.

IV. STATUS OF AMENDMENTS

No amendment has been filed after final.

V. SUMMARY OF THE CLAIMED SUBJECT MATTER

Subroutine calls allow frequently used code segments to be re-used from different points within a computer program. When a computer program reaches a point where it is desired to start execution of a subroutine, a call instruction is executed to redirect processing to the start of the subroutine. At the end of the subroutine, a return instruction is executed that returns processing control to the address within the program code immediately following the point from which the subroutine call was made.

Computer programs may use a mix of "native" instructions that may be directly executed by a processor core and "non-native" instructions that require interpretation into the form of native instructions prior to execution. Typically, native instructions can be executed more rapidly but have a lower code density and are more difficult to program than non-native instructions.

A common non-native subroutine may be called by either native calling code or non-native calling code. Since the non-native subroutine is executed in a non-native mode, when the subroutine has completed, it is required to determine whether a return is being made to a non-native calling program, in which case a mode change is not required,

or alternatively to a native calling program, in which case a mode change is required. This determination slows down processing.

The inventor observed that non-native subroutines are very often called from non-native calling programs and native subroutines are called from native calling programs with comparatively little intercalling between native and non-native routines. Thus, the mode checking usually indicates the mode is unchanged. But still the mode checking must be done for the relatively rare cases of an intercall between native and non-native code.

Having recognized the problems of intercalling between native and non-native instructions together with the inefficiencies of checking at each return instruction whether the call was made from native or non-native code, the designed separate and distinct instructions in the form of a return to non-native instruction and a return to native instruction whereby processing is respectively returned to a non-native instruction and a native instruction. In this way, flexibility to return either to a native instruction or to a non-native instruction may be provided without having to support the additional overhead of checking for the nature of the calling program each time a return instruction is encountered. Instead, specific types of return instruction are provided that effectively code into the program the nature of the return to be made. See the paragraph bridging pages 5 and 6 of the specification.

Figure 18 reproduced below schematically illustrates a first non-limiting example embodiment for providing subroutine return instructions that explicitly return either to native code or non-native code.

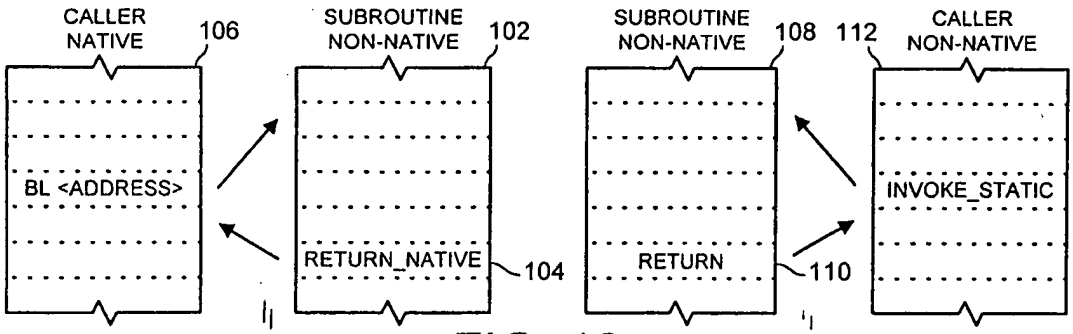


FIG. 18

A first subroutine 102 terminates with a return to native instruction 104. This non-native subroutine 102 is designed to be called from a native calling program 106 using a branch with link instruction (BL) to pass processing to the start of the non-native subroutine 102. The non-native subroutine 102 is also provided in a second form 108 which yields the same functionality except that a return to non-native instruction 110 is used rather than a return to native instruction 104. This second non-native subroutine is designed so as to be called from a non-native calling program 112. The explicit return to native instruction 104 and return to non-native instruction 110 avoid the need at the end of each subroutine 102, 108 to actively determine whether the calling program was native or non-native thereby speeding the overall subroutine execution by avoiding additional processing operations at its end. See the paragraph bridging pages 33-34.

The arrangement of Figure 18 has the disadvantage that two non-native subroutines 102 and 108 need to be provided which consumes memory resources. Figure 19 illustrates a refinement of the system of Figure 18. In Figure 19 a veneer non-native subroutine 114 is provided between the native calling program and the non-native subroutine. At the end of the non-native subroutine, a return to non-native instruction is

always executed and either returns processing to a non-native calling program or to a non-native veneer subroutine depending upon from where it was called. The provision of a common non-native subroutine saves memory space. The function of the non-native veneer subroutine is to receive the return call from the non-native subroutine and then pass control back to the native calling program via execution of a return to native instruction. Accordingly, the non-native veneer subroutine can be small. Page 34, lines 13-22.

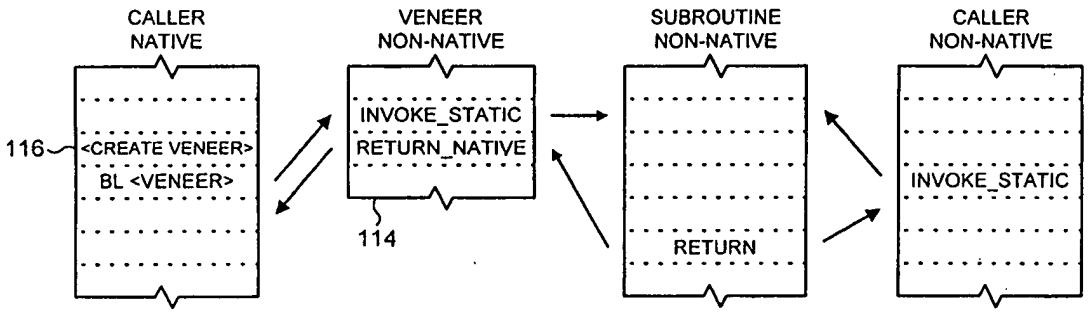


FIG. 19

The claimed processor core and processing means in claim 20 for executing native instructions can be for example the processor core 104 shown in Figure 5 or the processor core 120 shown in Figure 20. The claimed instruction translator and the translator means for interpreting non-native instructions in claim 20 can be for example the instruction translator 108 in Figure 5 or 122 in Figure 20. Both the processor and translator work with memory (means for storing in claim 24) referenced in Figure 5 and shown at 124 in Figure 20. See pages 13 and 35. Example instructions corresponding to the means for calling recited in claims 21 and 22 are shown in Figures 18 and 19.

Figure 20 schematically illustrates a non-limiting example data processing system 118 for performing the data processing operations illustrated in Figure 19.

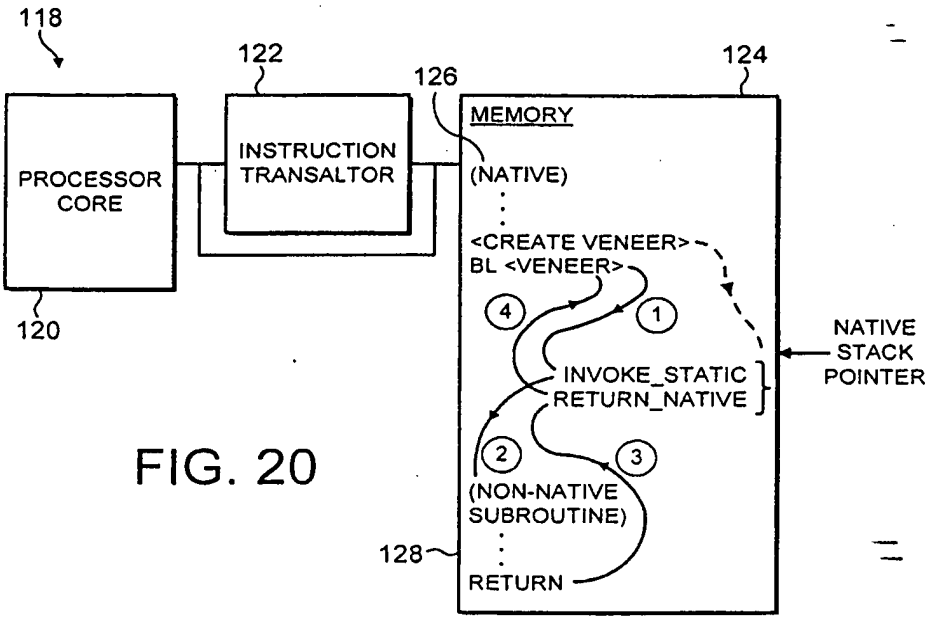


FIG. 20

The instruction translator 122 may provide hardware translation of Java bytecodes into native ARM instructions for simple Java bytecodes with more complicated Java bytecodes being translated using a software interpreter (not illustrated). Within the main memory 124, a native instruction program 126 is executed. Immediately prior to a subroutine call to a non-native subroutine, a veneer non-native subroutine is created in the stack memory portion of the main memory 124 as pointed to by the native stack pointer. After the veneer non-native subroutine has been created, it is branched to by a native branch instruction BL<veneer>. The veneer non-native subroutine then calls the non-native subroutine 128 which performs its desired data processing functions. At the end of the non-native subroutine 128, a return to non-native instruction is executed that returns processing to the veneer non-native subroutine within the stack memory region.

At the end of the veneer non-native subroutine, a return to native instruction is executed that finally returns processing to the native code 126. Page 35.

VI. GROUNDS OF REJECTION TO BE REVIEWED ON APPEAL

The anticipation rejection is requested for review on appeal.

VII. ARGUMENT

A. Legal Standard for Anticipation

To establish that a claim is anticipated, the Examiner must point out where each and every limitation in the claim is found in a single prior art reference. *Scripps Clinic & Research Found. v. Genentec, Inc.*, 927 F.2d 1565 (Fed. Cir. 1991). Every limitation contained in the claims must be present in the reference, and if even one limitation is missing from the reference, then it does not anticipate the claim. *Kloster Speedsteel AB v. Crucible, Inc.*, 793 F.2d 1565 (Fed. Cir. 1986). Guccione fails to satisfy this exacting standard.

B. The Guccione Reference

Java is a portable language and hardware independent. Thus, Java requires some mechanism for interfacing to either hardware or to non-Java code like device drivers and C or C++ libraries. Consider as an example a C programming language method incorporated into a Java program by defining an associated Java

class function to be of type "native." The Java interpreter must provide an interface to enable execution of the native method.

Guccione describes a method of calling native functions from Java. Three distinct native method interfaces are used including "stubs", Java native interface (JNI), and raw native interface (RNI). Because these three interfaces are incompatible, Guccione provides C interface code to build three different interface libraries, known as dynamically-linked libraries (DLL), corresponding to each one of the three native interfaces, as illustrated in Figure 1.

C. Guccione Does Not Disclose an Instruction Translator Responsive to a Return-to-a-Non-Native Instruction of a Non-Native Instruction Set to Return Processing to a Non-Native Instruction

Claims 1, 15, and 20 relate to an efficient approach where a non-native portion of program code returns either to a native calling program or a non-native calling program without the overhead of having to check on each return whether the calling program was a native program or a non-native program. This efficient approach is missing in Guccione.

The Examiner contends that Guccione teaches an instruction translator responsive to a return to a non-native instruction of the non-native instruction set to return processing to a non-native instruction at page 5 and Figure 5. Appellant disagrees.

The program code of Figure 5 is C program code which specifies C function prototype for the native method "PrintNum." Using the Examiner's correlation of "native" to the C code and non-native to Java code, the return instruction in the program code of Figure 5 is an instruction in the native (C) programming language and **not** a return instruction in the non-native (Java) programming language.

1. Figure 5 in Guccione is C Code—Not Java Code

The Examiner asserts in the final action on page 3 that because Figure 5 is the "stubs interface code, therefore, the return is a return instruction in the non-native (Java) programming language." The inventor has carefully reviewed Figure 5 and concluded that there is no doubt that the code in Figure 5 of Guccione is C code and not Java code. It is easy to see how the Examiner's misunderstanding arose because of the title: "The Java Stubs Interface Code." But "Java stubs" is a known term to those familiar with Java, and it refers to the set of stubs code written in C to allow a Java program to call native methods. There are a number of items throughout the Guccione paper, which clearly demonstrate that the code in Figure 5 is C code.

First, the code in Figure 5 would be syntactically incorrect as Java code. Java does not contain the keyword **struct**. To have the equivalent of a C struct in Java, you must instead have a **class** with no methods—just variables. Consider, for example, the Java equivalent of the following C struct

```
struct point {
  int x, y;
```

```
}
```

would be

```
class point {
  int x, y;
}
```

Because the code in Figure 5 uses the keyword **struct,** it would not be compiled by a Java compiler. Therefore, the code in Figure 5 must be C code.

Second, consider the sequences of Figures 2 through 5. Figure 2 shows "The Java code for the PrintNum example" with the following line:

```
public native int printNum(int num);
```

This line demonstrates that 'printNum' is a native C instruction. Figure 3 shows "The commands used to produce the stubs interface." The second command generates the 'PrintNum_stubs.c' source code file:

```
javah -stubs -o PrintNum_stubs.c PrintNum
```

This generates a 'Java stubs' file containing a call to the C function *PrintNum_printNum.*

Figure 5 then shows "The Java Stubs Interface Code" that interfaces between the 'Java stubs' C code generated as a result of the commands in Figure 3 and the C code in Figure 4. It is inconceivable that a person skilled in the programming art would use Java code to interface between two C functions.

As further evidence that the code in Figure 5 is C code, the inventor actually entered the code given in Guccione's Figure 2 and compiled it using the Java compiler, **javac**. He then used the commands in Figure 3 to generate the files, PrintNum_stubs.h and PrintNum_stubs.c. Because the inventor used a newer version of the Java

development system, he had to use the flag '-old' on the javah command line to generate the stubs corresponding to SUN JDK 1.0.

The following are the files generated by the PrintNum_stubs.h and PrintNum_stubs.c.

PrintNum_stubs.h

```
/* DO NOT EDIT THIS FILE - it is machine generated */
#include <native.h>
/* Header for class PrintNum */

#ifndef _Included_PrintNum
#define _Included_PrintNum

#pragma pack(4)

typedef struct ClassPrintNum {
    char PAD; /* ANSI C requires structures to have a
least one member */
} ClassPrintNum;
HandleTo(PrintNum);

#pragma pack()

#ifdef __cplusplus
extern "C" {
#endif
extern int32_t PrintNum_printNum(struct HPrintNum
*,int32_t);
#ifdef __cplusplus
}
#endif
#endif
```

This clearly shows the definition of **PrintNum_printNum** as a C function. So the code in Figure 5 must therefore be C code.

PrintNum_stubs.c

```
/* DO NOT EDIT THIS FILE - it is machine generated */
#include <StubPreamble.h>

/* Stubs for class PrintNum */
/* SYMBOL: "PrintNum/printNum(I)I",
Java_PrintNum_printNum_stub */
__declspec(dllexport)stack_item
*Java_PrintNum_printNum_stub(stack_item *_P_,struct
execenv *_EE_) {
     extern int32_t PrintNum_printNum(void *,int32_t);
     _P_[0].i =
PrintNum_printNum(_P_[0].p,((_P_[1].i)));
     return _P_ + 1;
}
```

This clearly shows **PrintNum_printNum** being called as a C function. Accordingly, the code in Figure 5 must be C code.

2. Guccione Does Not Teach the Claimed Translation

The Examiner argues that Guccione's DLL is the claimed instruction translator because the DLL allows a native method to be incorporated in a non-native computer program. This argument is not persuasive.

A DLL (Dynamic Link Library) simply provides a library of native methods, or functions, which can be called by Java (or by C). Figure 4 provides an example of such a function. The DLLs described by Guccione also incorporate the "Java Stubs" and the C interface code shown in Figure 5. That is all the DLL provides—a library of functions. There is no translation of non-native instructions to native instructions involved.

The claimed instruction translator interprets non-native instructions into native instructions for execution. In contrast, the DLLs described by Guccione don't do this.

The DLL contains only native instructions and does not perform any translation or interpretation of non-native instructions. In fact, it is technically impossible for the DLL to perform any translation or interpretation, since it does not have access to the non-native instructions.

The DLL does not even known whether it was called from native or non-native code. In this regard, Guccione says after Figure 4: "Note that this code is independent of Java, and can compile on its own and may also be used to provide a library interface to other C code." If the DLL can be called from other C code, then the DLL cannot be performing any sort of instruction translation on non-native code since there is no non-native code to translate. In short, the DLL of Guccione is not "operable to interpret non-native instruction of a non-native instruction set into native instructions," as recited in independent claim 1.

D. Guccione Does Not Disclose an Instruction Translator Responsive to a Return-to-Native Instruction of a Non-Native Instruction Set to Return Processing to a Native Instruction

The Examiner admits on page 4 of the final action that Guccione does not explicitly describe an instruction translator "responsive to a return to native instruction of said non-native instruction set to return processing to a native instruction" (quoted from claim 1). Nonetheless, the Examiner argues that the stubs are "created for both directions so the native instruction and non-native instruction exchange some information. Therefore, the opposite direction...must be disclosed otherwise the stubs would not work at all." Appellant disagrees with strained inherency position. *Id.*

The background section of this application explains that it is known to call a non-native subroutine from either native calling code or non-native calling code, and at the end of the subroutine, return to the calling code using a return instruction. But the problem is that the non-native subroutine must determine whether the return is to a native calling program or to a non-native calling program. This method is inefficient because of the additional overhead associated with checking for the type of calling program each time a return instruction is encountered.

Thus, even if it were, as the Examiner suggests, inherent in Guccione that there is *some method* of returning from a non-native code subroutine to native code via the Java stubs interface, it certainly does not follow that Guccione discloses the *particular* method or apparatus of return as specified in the independent claims. No where does Guccione disclose using two different return instructions—a return to non-native instruction of the non-native instruction set and return to native instruction of the non-native instruction set—that effectively code into the program the nature of the return to be made.

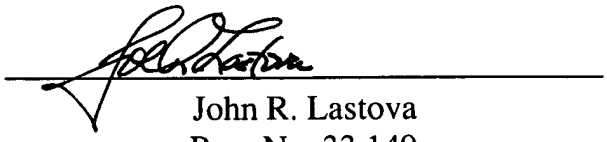
Guccione's stubs code enables a native method to be incorporated in a non-native computer program. According, there is no need to provide a special-purpose, return to native instruction as specified, for example, by claim 1 integer (iv). Once Guccione's C method has executed, the program execution will invariably return to non-native code thereby obviating any need to provide a return to native instruction.

VIII. CONCLUSION

Lacking multiple features from the independent claims 1, 15, and 20, the Board should reverse the outstanding rejections.

Respectfully submitted,

NIXON & VANDERHYE P.C.

By: 

John R. Lastova
Reg. No. 33,149

JRL/sd
Appendix A - Claims on Appeal

IX. CLAIMS APPENDIX

1. Apparatus for processing data, said apparatus comprising:

(i) a processor core operable to execute native instructions of a native instruction set; and

(ii) an instruction translator operable to interpret non-native instructions of a non-native instruction set into native instructions for execution by said processor core; wherein

(iii) said instruction translator is responsive to a return to non-native instruction of said non-native instruction set to return processing to a non-native instruction; and

(iv) said instruction translator is responsive to a return to native instruction of said non-native instruction set to return processing to a native instruction.

2. Apparatus as claimed in claim 1, wherein said instruction translator is a hardware based instruction translator.

3. Apparatus as claimed in claim 1, wherein said instruction translator is a software based interpreter.

4. Apparatus as claimed in claim 1, wherein said instruction translator is a combination of a hardware based instruction translator and a software based interpreter.

5. Apparatus as claimed in claim 1, wherein said non-native instructions are Java Virtual Machine instructions.

6. Apparatus as claimed in claim 1, wherein a non-native subroutine is called from native code via a non-native veneer subroutine, such that, upon completion of said

non-native subroutine, a return to non-native instruction can be used to return processing to said non-native veneer subroutine with a return to native instruction within said non-native veneer subroutine serving to return processing to said native code.

7. Apparatus as claimed in claim 6, wherein said non-native subroutine is also called from non-native code.

8. Apparatus as claimed in claim 6, wherein said non-native veneer subroutine is dynamically created when said non-native subroutine is called from native code.

9. Apparatus as claimed in claim 8, wherein said non-native veneer subroutine is created stored within a stack memory area used by native code operation.

10. Apparatus as claimed in claim 1, wherein said instruction translator is responsive to a plurality of types of return to non-native instruction.

11. Apparatus as claimed in claim 10, wherein said plurality of types of return to non-native instruction are operable to return with respective different types of return value.

12. Apparatus as claimed in claim 11, wherein said plurality of different types of return value include one or more of:

(i) a 32-bit integer return value;

(ii) a 64-bit integer return value;

(iii) an object reference return value;

(iv) a single precision floating point return value;

(v) a double precision floating point return value; and

(vi) a void return value having no value.

13. Apparatus as claimed in claim 1, wherein said instruction translator is responsive to a plurality of types of return to native instruction.

14. Apparatus as claimed in claim 13, wherein said plurality of types of return to native instruction are operable to return with respective different types of return value.

15. A method of processing data, said method comprising the steps of:

(i) executing native instructions of a native instruction set using a processor core; and

(ii) interpreting non-native instructions of a non-native instruction set into native instructions for execution by said processor core; wherein

(iii) in response to a return to non-native instruction of said non-native instruction set, returning processing to a non-native instruction; and

(iv) in response to a return to native instruction of said non-native instruction set, returning processing to a native instruction.

16. A computer program product carrying a computer program for controlling a data processing apparatus in accordance with the method of claim 15.

20. Apparatus for processing data, comprising:

processing means for executing native instructions of a native instruction set; and

translator means for interpreting non-native instructions of a non-native instruction set into native instructions for execution by said processing means being responsive to a return to non-native instruction of said non-native instruction set to return processing to a non-native instruction, and responsive to a return to native instruction of said non-native instruction set to return processing to a native instruction.

21. Apparatus as claimed in claim 20, further comprising:

means for calling a non-native subroutine from native code via a non-native veneer subroutine, such that, upon completion of said non-native subroutine, a return to non-native instruction can be used to return processing to said non-native veneer subroutine with a return to native instruction within said non-native veneer subroutine serving to return processing to said native code.

22. Apparatus as claimed in claim 21, further comprising:

means for calling said non-native subroutine from non-native code.

23. Apparatus as claimed in claim 21, further comprising:

means for dynamically creating said non-native veneer subroutine when said non-native subroutine is called from native code.

24. Apparatus as claimed in claim 23, further comprising:

means for storing said created non-native veneer subroutine within a stack memory area used by native code operation.

25. Apparatus as claimed in claim 20, wherein said translator means is responsive to a plurality of types of return to non-native instruction.

26. Apparatus as claimed in claim 25, wherein said plurality of types of return to non-native instruction are operable to return with respective different types of return value.

27. Apparatus as claimed in claim 20, wherein said instruction translator is responsive to a plurality of types of return to native instruction.

28. Apparatus as claimed in claim 27, wherein said plurality of types of return to native instruction are operable to return with respective different types of return value.

X. EVIDENCE APPENDIX

There is no evidence appendix.

XI. RELATED PROCEEDINGS APPENDIX

There is no related proceedings appendix.